SSI module

SSI or Server Side Includes, is actually a sort of server-side programming language interpreted by Nginx. Its name originates from the fact that the most-used functionality of the language is the `include` command. Back in the 1990s, such languages were employed in order to render web pages dynamically, from simple static `.html` files with client-side scripts to complex pages with server-processed instructions. Within the HTML source code, webmasters could now insert server-interpreted directives, which would then lead the way to much more advanced pre-processors such as PHP or ASP.

The most famous illustration of SSI is the *quote of the day* example. In order to insert a new quote every day at the top of each page of their website, webmasters would have to edit out the HTML source of every page of the site, updating the old quote manually. With Server Side Includes, a single command suffices to simplify the task:

```
<html>
<head><title>My web page</title></head>
<body>
  <h1>Quote of the day: <!--# include file="quote.txt" -->
  </h1>
</body>
</html>
```

All you would have to do to update the quote is to edit the contents of the `quote.txt` file. Automatically, all the pages would show the updated quote. As of today, most of the major web servers (Apache, IIS, Lighttpd, and so on) support Server Side Includes.

Module directives and variables

Having directives inserted within the actual content of the files that Nginx serves raises one major issue—what files should Nginx parse for the SSI commands? It would be a waste of resources to parse binary files such as images (`.gif`, `.jpg`, and `.png`) or other kinds of media, since they are unlikely to contain any SSI commands. You need to make sure to configure Nginx correctly with the directives introduced by this module:

Directive	Description
`ssi` Context: `http`, `server`, `location`, `if`	Enables parsing files for SSI commands. Nginx only parses the files corresponding to the MIME types selected with the `ssi_types` directive. Syntax: `on` or `off` Default value: `off` `ssi on;`

`ssi_types`
Context: `http`, `server`, `location`
	Defines the MIME file types that should be eligible for SSI parsing. The `text/html` type is always included.
	Syntax:
	`ssi_types type1 [type2] [type3...];`
	`ssi_types *;`
	Default value: `text/html`
	`ssi_types text/plain;`

`ssi_silent_errors` Context: `http`, `server`, `location`	Some SSI commands may generate errors; in that case, Nginx outputs a message at the location of the command—'an error occurred while processing the directive'. Enabling this option silences Nginx and the message does not appear. Syntax: `on` or `off` Default value: `off` `ssi_silent_errors off;`
`ssi_value_length` Context: `http`, `server`, `location`	SSI commands have arguments that accept a value (for example, `<!--# include file="value" -->`). This parameter defines the maximum length accepted by Nginx. Syntax: Numeric Default: `256` (characters) `ssi_value_length 256;`

Directive	Description
`ssi_ignore_recycled_buffers` Context: `http`, `server`, `location`	When set to `on`, this directive prevents Nginx from making use of the recycled buffers. Syntax: `on` or `off` Default: `off`
`ssi_min_file_chunk` Context: `http`, `server`, `location`	If the size of a buffer is greater than `ssi_min_file_chunk`, data is stored in a file and then sent via `sendfile`. In other cases, it is transmitted directly from the memory. Syntax: Numeric value (size) Default: `1,024`
`ssi_last_modified` Context: `http`, `server`, `location`	If set to `off`, Nginx removes the `Last-modified` header from the original response during SSI processing in order to increase the caching likeliness. The `Last-modified` date is likely to change often due to dynamically generated elements contained in the response, rendering it non-cacheable. Syntax: `on` or `off` Default: `off`

A quick note regarding possible concerns about the SSI engine resource usage—by enabling the SSI module at the `location` or `server` block level, you enable parsing of at least all `text/html` files (pretty much any page to be displayed by the client browser). While the Nginx SSI module is efficiently optimized, you might want to disable parsing for the files that do not require it.

Firstly, all your pages containing the SSI commands should have the `.shtml` (Server HTML) extension. Then, in your configuration at the `location` block level, enable the SSI engine under a specific condition. The name of the served file must end with `.shtml`:

```
server {
    server_name website.com;
    location ~* \.shtml$ {
        ssi on;
    }
}
```

On one hand, all HTTP requests submitted to Nginx will go through an additional regular expression pattern matching. On the other hand, static HTML files or files to be processed by other interpreters (`.php`, for instance) will not be parsed unnecessarily.

Finally, the SSI module enables two variables:

- `$date_local`: Returns the current time according to the current system time zone
- `$date_gmt`: Returns the current GMT time, regardless of the server time zone

SSI commands

Once you have the SSI engine enabled for your web pages, you are ready to start writing your first dynamic HTML page. Again, the principle is simple—design the pages of your website using the regular HTML code, inside which you will insert the SSI commands.

These commands respect a particular syntax—at first sight, they look like regular HTML comments: `<!-- A comment -->`, and that is the good thing about it—if you accidentally disable SSI parsing of your files, the SSI commands do not appear on the client browser; they are only visible in the source code as actual HTML comments. The full syntax is as follows:

```
<!--# command param1="value1" param2="value2" … -->
```

File includes

The main command of the Server Side Include module is, obviously, the `include` command. It can be used in two different fashions.

First, you are allowed to make a simple file include:

```
<!--# include file="header.html" -->
```

This command generates an HTTP sub-request to be processed by Nginx. The body of the response that was generated is inserted instead of the command itself.

The second possibility is to use the `include virtual` command:

```
<!--# include virtual="/sources/header.php?id=123" -->
```

This also sends a sub-request to the server; the difference lies in the way that Nginx fetches the specified file (when using `include file`, the `wait` parameter is automatically enabled). Indeed, two parameters can be inserted within the `include` command tag. By default, all SSI requests are issued simultaneously, in parallel. This can cause slowdowns and timeouts in case of heavy loads. Alternatively, you can use the `wait="yes"` parameter to specify that Nginx should wait for the completion of the request before moving on to other includes:

```
<!--# include virtual="header.php" wait="yes" -->
```

If the result of your `include` command is empty or if it triggered an error (`404`, `500`, and so on), Nginx inserts the corresponding error page with its HTML: `<html>[...]404 Not Found</body></html>`. The message is displayed at exactly the same place where you inserted the `include` command. If you wish to revise this behavior, you have the option to create a named block. By linking the block to the `include` command, the contents of the block will show at the location of the `include` command tag in case an error occurs:

```
<html>
<head><title>SSI Example</title></head>
<body>
<center>
  <!--# block name="error_footer" -->Sorry, the footer file was not
found.<!--# endblock -->
  <h1>Welcome to nginx</h1>
  <!--# include virtual="footer.html" stub="error_footer" -->
</center>
</body>
</html>
```

The result as output in the client browser is shown as follows:

As you can see, the contents of the `error_footer` block were inserted at the location of the `include` command, after the `<h1>` tag.

Working with variables

The Nginx SSI module also offers the option of working with variables. Displaying a variable (in other words, inserting the variable value into the final HTML source code) can be done with the `echo` command:

```
<!--# echo var="variable_name" -->
```

The command accepts the following three parameters:

- `var`: The name of the variable that you want to display, for example, `REMOTE_ADDR` to display the IP address of the client.
- `default`: A string to be displayed in case the variable is empty. If you don't specify this parameter, the output is (`none`).
- `encoding`: Encoding method for the string. The accepted values are `none` (no particular encoding), `url` (encode text like a URL—a blank space becomes `%20`, and so on), and `entity` (uses HTML entities: `&` becomes `&`).

You may also affect your own variables with the `set` command:

```
<!--# set var="my_variable" value="your value here" -->
```

The `value` parameter is itself parsed by the engine; as a result, you are allowed to make use of the existing variables:

```
<!--# echo var="MY_VARIABLE" -->
<!--# set var="MY_VARIABLE" value="hello" -->
<!--# echo var="MY_VARIABLE" -->
<!--# set var="MY_VARIABLE" value="$MY_VARIABLE there" -->
<!--# echo var="MY_VARIABLE" -->
```

The following is the output that Nginx displays for each of the three `echo` commands from the preceding example:

```
(none)
hello
hello there
```

Conditional structure

The following set of commands allow you to include text or other directives depending on a condition. The conditional structure can be established with the following syntax:

```
<!--# if expr="expression1" -->
[…]
<!--# elif expr="expression2" -->
```

```
[…]
<!--# else -->
[…]
<!--# endif -->
```

The expression can be formulated in three different ways:

- Inspecting a variable: `<!--# if expr="$variable" -->`. Similar to the `if` block in the Rewrite module, the condition is true if the variable is not empty.
- Comparing two strings: `<!--# if expr="$variable = hello" -->`. The condition is true if the first string is equal to the second string. Use `!=` instead of `=` to revert the condition (the condition is true if the first string is not equal to the second string).
- Matching a regular expression pattern: `<!--# if expr="$variable = /pattern/" -->`. Note that the pattern must be enclosed within `/` characters, otherwise it is considered to be a simple string (for example, `<!--# if expr="$MY_VARIABLE = /^/documents//" -->`). Similar to the comparison, use `!=` to negate the condition. The captures in regular expressions are supported.

The content that you insert within a condition block can contain regular HTML code or additional SSI directives with one exception—you cannot nest `if` blocks.

Configuration

Last and probably the least (for once) of the SSI commands offered by Nginx is the `config` command. It allows you to configure two simple parameters.

First, the message that appears when the SSI engine faces an error related to malformed tags or invalid expressions. By default, Nginx displays `[an error occurred while processing the directive]`. If you want it to display something else, enter the following:

```
<!--# config errmsg="Something terrible happened" -->
```

Additionally, you can configure the format of the dates that are returned by the `$date_local` and `$date_gmt` variables using the `timefmt` parameter:

```
<!--# config timefmt="%A, %d-%b-%Y %H:%M:%S %Z" -->
```

The string that you specify here is passed as the format string of the `strftime` C function. For more information about the arguments that can be used in the format string, please refer to the documentation of the `strftime` C language function at `http://www.opengroup.org/onlinepubs/009695399/functions/strftime.html`.